Terms used **compact garbage collection tables** Found **39,815** of **820,928**

Sort results by: relevance
Display results: expanded form
Save results to a Binder
Search Tips
☐ Open results in a new window

Try an Advanced Search
Try this search in The Digital Library

Results 1 - 20 of 200 Result page: **1** 2 3 4 5 6 7 8 9 10 next
Best 200 shown
Relevance scale

1 Compact garbage collection tables
David Tarditi
October 2000 **ACM SIGPLAN Notices , Proceedings of the second international symposium on Memory management**, Volume 36 Issue 1
Full text available: pdf(958.92 KB) Additional Information: full citation, abstract, index terms

Garbage collection tables for finding pointers on the stack can be represented in 20-25% of the space previously reported. Live pointer information is often the same at many call sites because there are few pointers live across most call sites. This allows live pointer information to be represented compactly by a small index into a table of descriptions of pointer locations. The mapping from program counter values to those small indexes can be represented compactly using several techniques. T ...

2 Comparison of Compacting Algorithms for Garbage Collection
Jacques Cohen, Alexandru Nicolau
October 1983 **ACM Transactions on Programming Languages and Systems (TOPLAS)**, Volume 5 Issue 4
Full text available: pdf(1.11 MB) Additional Information: full citation, references, citings, index terms

3 Garbage Collection of Linked Data Structures
Jacques Cohen
September 1981 **ACM Computing Surveys (CSUR)**, Volume 13 Issue 3
Full text available: pdf(2.32 MB) Additional Information: full citation, references, citings, index terms

4 Compact list representation: definition, garbage collection, and system implementation
Wilfred J. Hansen
September 1969 **Communications of the ACM**, Volume 12 Issue 9
Full text available: pdf(1.19 MB) Additional Information: full citation, abstract, references, citings, index terms

Compact lists are stored sequentially in memory, rather than chained with pointers. Since this is not always convenient, the Swym system permits a list to be chained, compact, or any combination of the two. A description is given of that list representation and the operators implemented (most are similar to those of LISP 1.5). The system garbage

Compact Garbage Collection Tables

David Tarditi
Microsoft Research
Redmond, WA 98052, USA
dtarditi@microsoft.com

ABSTRACT

Garbage collection tables for finding pointers on the stack can be represented in 20-25% of the space previously reported. Live pointer information is often the same at many call sites because there are few pointers live across most call sites. This allows live pointer information to be represented compactly by a small index into a table of descriptions of pointer locations. The mapping from program counter values to those small indexes can be represented compactly using several techniques. The techniques all assign numbers to call sites and use those numbers to index an array of small indexes. One technique is to represent an array of return addresses by using a two-level table with 16-bit offsets. Another technique is to use a sparse array of return addresses and interpolate the exact number via disassembly of the executable code.

1. INTRODUCTION

A copying garbage collector has to find all memory locations on the call stack that point to heap-allocated data. One successful approach to this problem is for a compiler to generate tables that the garbage collector uses to traverse the call stack and find the locations. A drawback of the approach is that the tables can be large relative to executable code size. Diwan et al. report tables that are 16% of VAX code size [5] and Stichnoth et al. report tables that are 20% of 80x86 code size [12]. When code size is important, the large tables are a significant cost to pay for garbage collection support.

This paper describes a design for compact garbage collection tables that can be accessed and decoded quickly. The tables are on average 3.6% of code size for programs compiled with an optimizing whole-program Java compiler for the 80x86. The design is based on the empirical observation that live pointer information is often the same at many call sites. This is because most call sites have few pointers live across them. The live pointer information can be represented compactly using a small index into a table of pointer location descriptions. Furthermore, the number of indexes can be reduced by using compiler optimizations that pack pointer-containing locations close together in stack frames. The design also uses several techniques for compactly mapping program counter values to those small indexes. The techniques all assign numbers to call sites and use those numbers to index an array of small indexes. One technique is to assign adjacent call sites that have the same live pointer information the same number [5]. Another technique is to represent an array of return addresses by using a two-level table with 16-bit offsets. A final technique is to use a sparse array of return addresses and interpolate the number by disassembling the executable code.

The paper studies the properties of 20 benchmarks compiled using a whole-program Java compiler. The benchmarks include the SPEC JVM98 benchmarks [4]. It also uses those benchmarks to evaluate the design and compare various techniques for mapping program counter values to small indexes.

Diwan et. al [5] describe garbage collection tables for optimized code. Their primary focus is not on building compact tables, although they address the issue of building tables with reasonable sizes. They describe how to support derived interior pointers that are produced by optimizations such as strength reduction and induction variable elimination. The work described here focuses on building compact tables and does not address support for derived interior pointers. It also studies benchmarks that are one to four orders of magnitude larger than the benchmarks studied by Diwan et. al. Empirical observations of these larger benchmarks show effects that cannot be observed with smaller benchmarks, such as live pointer information being the same at many call sites.

Stichnoth et al. [12] describe tables that allow garbage collection to begin at almost every instruction. For multi-threaded programs, it reduces the potential start-up latencies of garbage collection at the expense of larger tables. It is unnecessary for single-threaded programs. In contrast, the design described in this paper allows garbage collection to begin only at specified program points. This produces smaller tables, but it may increase the start-up latency of garbage collection. The latency can be bounded by making the beginnings of extended basic blocks [2] or the tops of loops [5] garbage collection points. An explicit check can be added or the language runtime system can insert a breakpoint to halt thread execution when those points are reached.

Stichnoth et al. also compress the binary representation of their tables by using Huffman encoding. The design described here relies on explicit sharing instead of a generic compression algorithm.

The remainder of the paper is organized as follows. Section 2 describes the basic structure of the tables, how the runtime system interprets the tables, and how the tables are generated. Section 3 describes how to reduce the size of the mapping from program counter values to indexes. Section 4 describes the testbed used to evaluate the tables. Section 5 measures the size of the tables, the sharing that occurs, and the effects of improvements.

2. USING SMALL INDEXES TO REPRESENT LIVE POINTER INFORMATION

This section describes how the garbage collection tables use small indexes to represent the descriptions of pointer locations and how the tables are interpreted and generated. It also describes compiler optimizations that reduce the number of indexes and the format of the descriptions of pointer locations for the 80x86.

Garbage collection is assumed to occur only upon a call from a thread into the runtime system, that is, at defined program points. For single-threaded programs those points are allocation sites. For multi-threaded programs, those points are allocation sites or system calls that cause a thread to suspend its execution. The problem of finding pointer locations becomes the problem of mapping the return address for each call back to a description of the locations of pointers that are live when the call was made.

2.1 Table organization

Figure 1 shows the organization of the tables. The *call site table* and the *descriptor index table* map program counter values to small indexes. More compact representations of the call site table are discussed in Section 3. The *call site table* is a sorted array of 32-bit return addresses that maps call sites to their numbers. The *descriptor index table* maps call site numbers to indexes; it is an array of indexes. The *descriptor table* contains descriptions of the locations of pointers.

Descriptors have two forms. The compact form encodes the pointer locations into a 32-bit word. The escape form is a pointer to a variable-length record that describes pointer locations. The least-significant bit of a descriptor is used to distinguish between the the two forms. The details of the formats are described in Section 2.5.

2.2 Table interpretation

Figure 2 show pseudo-code for walking the stack and processing stack frames. The `walkStack` function takes the frame pointer for the youngest Java stack frame and the return address where execution will resume using that frame. It then walks the stack frames until it finds a non-Java stack frame. For each stack frame, it uses a binary search of the call site table to find the call site number. It then uses the call site number to look up the descriptor index in the descriptorIndexTable and uses the descriptor index to look up the actual descriptor. The function processFrame (not shown) decodes the descriptor and forwards live heap pointers for the call.

2.3 Table construction

Table construction starts with a list of all call sites where garbage collection may occur. The list is ordered by call site addresses. For each call site, the return address where execution will resume and live pointer information is recorded. Live pointer information includes the set of stack locations and callee-save registers that contain pointers live across the call site and the set of callee-saved registers that are unchanged on all paths to the call site.

First, the set of descriptors is computed. The list of call sites is copied and sorted lexicographically using the live pointer information. The sorted list is scanned and every time the live pointer information changes, a new index is assigned and the live pointer information is added to the list of descriptors. The descriptor index is recorded for the run of call sites with the same information.

Finally, the three tables are generated. The call site table is formed by creating an array from the return addresses stored in the list of call sites The descriptor index table is formed by creating an array from the descriptor indices stored in the list of call site sets. The descriptor table is created by converting the list of descriptors to an array of 32-bit integers encoding the descriptors.

2.4 Compiler optimizations to reduce the number of indexes

The number of distinct descriptions of pointer locations (and thus indexes) can be reduced by improving the spatial locality of pointers on the stack. First, the stack frame is divided into pointer-containing and non-pointer containing sections. The pointers are placed in the stack slots closest to the beginning of the local variable area. Second, stack slots are colored so that they are reused for variables with disjoint lifetimes. This is important for large functions that have many live variables.

2.5 Describing pointer locations for the 80x86

The descriptor formats depend on the register usage, the calling convention, and the stack layout. The 80x86 has 8 general-purpose registers, two of which are used as the stack pointer and frame pointer. The remaining registers are divided into three callee-save registers and three caller-save registers. Arguments are passed in the caller-save registers and on the stack. The formats currently assume that all functions use frame pointers.

Figure 3 shows the layout of a stack frame. The stack grows downward toward lower addresses. The frame pointer points to the saved frame pointer of the caller. Argument values are at positive offsets from the frame pointer and local variables are at negative offsets. Callee-save registers are saved in the memory locations immediately below the old value of the frame pointer.

Figure 4 shows the format of a compact descriptor. Each compact descriptor has five bitfields. The first 3-bit field describes what callee-save registers were saved on the stack at function entry. The second 6-bit field describes the usage of the callee-save registers across a call. The third 4-bit field describes the highest 32-bit word that contains a live pointer in the argument section of the stack. The bitfield is zero if there are no live pointer arguments. The fourth bitfield is a bitmask that indicates what argument stack locations contain live pointers. Its length is variable and determined by the third bitfield. The fifth bitfield is a bitmask that uses the remaining bits and indicates what local variable stack locations contain pointers.

Escape descriptor records have three variants that de-

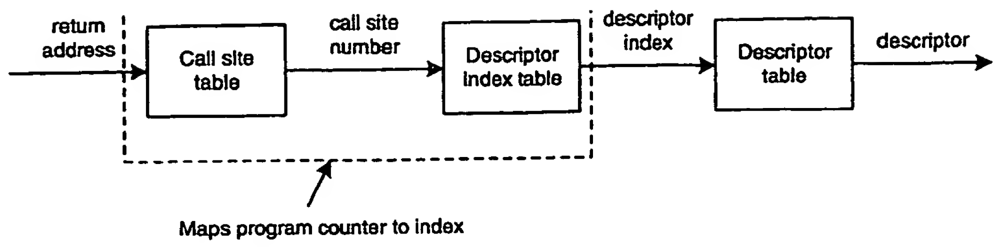

Figure 1: Table organization

```
// , fp = frame pointer, ra = current return address
walkStack(char *fp, char *ra) {
   while (ra is for a Java function) {
      find the call site number n of ra using binary search
      index = descriptorIndexTable[n]
      descriptor = descriptorTable[index]
      processFrame(descriptor,fp);
      a = *(fp+4)
      fp = *(fp)
   }
}
```

Figure 2: Pseudo-code for walking the stack

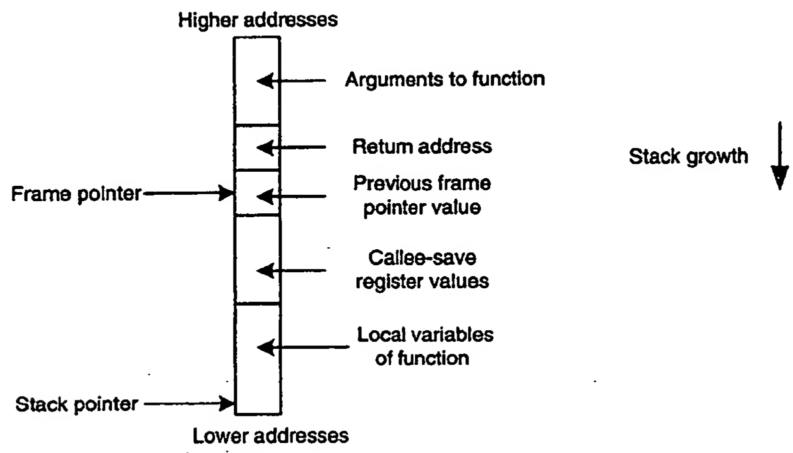

Figure 3: The layout of a stack frame.

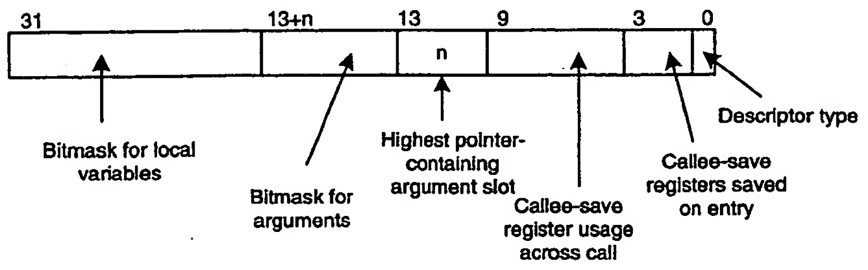
31
13+n
13
9
3
0
n
Bitmask for local variables
Bitmask for arguments
Highest pointer-containing argument slot
Callee-save register usage across call
Callee-save registers saved on entry
Descriptor type

Figure 4: The format of a compact descriptor

scribe 8, 16, and 32-bit offsets respectively. The first field describes the variant type and also includes the bitfields for the callee-save registers. The second field describes the number of pointers on the stack and the rest of the record contains the frame pointer offsets of the pointer locations. All pointers are aligned, so the offsets are in units of 4 bytes.

3. COMPACT MAPPINGS FROM PROGRAM COUNTER VALUES TO INDEXES

The call site table described in the previous section was an array of 4-byte return addresses. This section describes three techniques for reducing the size of the call site table.

3.1 Grouping adjacent call sites into sets

Diwan et al. [5] observe that adjacent call sites often have the same live pointer information. These adjacent call sites, along with call sites that cannot garbage collect, can be grouped into sets. The call site table no longer maps individual return addresses to call site numbers. Instead, it maps ranges of program counter values to call site set numbers.

3.2 Using a 2-level table with 16-bit offsets

The call site table can be implemented as a 2-level table with 16-bit offsets. Each 64K segment of the instruction address space has its own table that keeps the offsets of call sites within the segment. The starting call site number is also recorded for each segment.

If call sites are reasonably dense, this technique roughly halves the call site table size, replacing 32 bits of information with 16 bits of information. A drawback of the technique is that the compiler must know the offset of each label from the base of the code segment to calculate the 64K segment for the label and the 16-bit offset within that segment. For the 80x86, which has variable length instructions, the Marmot compiler uses it own assembler and generates object files directly when using a 2-level call site table.

3.3 Using a sparse table and interpolating via disassembly

Another approach to reducing the call site table size is based on the observation that executable code itself contains some of the information in the table. Specifically, the number of a call site can be calculated by disassembling the executable code from the start of the code to the current return address, assuming that no data has been intermixed with the code. From this perspective, tables accelerate the disassembly process and allow the disassembly process to skip data that is intermixed with code. The tables also support grouping call sites into sets as described in Section 3.1.

Three tables can be used to interpolate call site set numbers by disassembling the executable code. The first table is a sparse call site table that has a subset of the entries in the original table. For example, the sparse table could have $1/n$ of the entries of the original table, where n is a compiler parameter. The second table is a back-mapping table that maps each sparse entry to its call site number and a call site set number. The third table is a bitmask that indicates which call sites are the beginnings of call site sets.

The calculation of the call site set number for a given return address proceeds in two steps. First, the sparse call site table is searched to find the entry that is closest to the return address and that precedes it. Second, the exact call site set number is determined by interpolation. The code is disassembled from the entry address to the return address and the number of call site sets between the two points is counted. The call site number of the sparse entry is used to index into the call site bitmask to figure out what call sites to count. The number of call site sets is then added to the call site set number of the sparse entry.

The back-mapping table is broken into segments that map sparse call entries to 16-bit offsets for call site and call site set numbers. The offsets are added to base call site and call site set numbers for the segment. On average, each sparse entry needs only 4 bytes of information in the table.

The disassembler only needs to recognize call instructions and calculate instruction lengths. The runtime system code to do this for the 80x86 consists of 120 lines of table-driven C code and 300 lines of table initialization code, even though the 80x86 has a highly irregular instruction format with variable length instructions. The disassembler would be simpler for a RISC architecture where all instructions are the same length.

The cost of disassembling code can be significant on the 80x86 because the irregular instruction format means that code must be examined one byte at a time. A 16-element LRU cache is used to amortize the cost of disassembling code; the cache maps return addresses to their call site set numbers.

An advantage of the sparse call site table over the 2-level table is that the compiler no longer needs to include an assembler because it does not need to calculate precise offsets. A disadvantage of the sparse call site table is that a debugger may confuse the disassembly process when it inserts a breakpoint instruction. The breakpoint instruction may replace a call instruction or, on a machine like the 80x86 with variable length instructions, overwrite part of an instruction. To avoid confusing the disassembler, the debugger must inform the disassembler about any changes that it makes to the executable code.

4. EXPERIMENTAL TESTBED

This section describes the experimental testbed. It consists of an optimizing compiler, a set of benchmarks, and the machine used for timing measurements.

4.1 The Marmot optimizing compiler for Java

Marmot [6] is a whole-program, optimizing Java compiler that was developed as a research platform. It produces 80x86 executables and includes standard scalar optimizations such as constant and copy propagation, common subexpression elimination, dead-assignment elimination, loop invariant code motion, induction variable elimination, strength reduction, and inlining of small, statically called functions. It also includes basic object-oriented optimizations such as virtual call rebinding based on class hierarchy analysis, null check elimination, type test elimination, and removal of uninvoked methods and advanced optimizations such as stack allocation of objects [7], array bounds check elimination, and the elimination of unnecessary synchronization operations [11].

All programs were compiled using a generational garbage collector with a write barrier. A card-marking write barrier was used by default because a sequential store buffer (SSB) write barrier may increase the size of tables (see Section 5.8).

4.2 The benchmarks

Table 1 lists the benchmarks. The benchmarks are divided into three groups. The SPEC JVM98 group includes the programs from the SPEC JVM98 benchmark suite [4]. The MiscJava group includes various Java programs. It includes the Marmot compiler, which is the largest program used in this study. The IMPACT group includes C programs translated to Java by the IMPACT/NET project [9]. Several of the original C programs came from the SPEC95 suite [3].

Section 5 focuses on four benchmarks that illustrate a range of results: 201_compress, 202_jess, wc, and Marmot. 201_compress and 202_jess are typical of the SPEC JVM98 and Misc programs. Wc is typical for the IMPACT programs. Marmot has the largest table size relative to code size of all the benchmarks.

4.3 Hardware platform

Benchmarks were run on an otherwise unloaded 300 MHz dual Pentium II (x86 Family 6 Model 5) processor Gateway 2000 E-5000 running Windows NT 2000. The machine was disconnected from the local area network. High-resolution on-chip cycle counters were used to measure the time spent garbage collecting; entry cycle counts were subtracted from exit cycle counts. The benchmarks were run repeatedly until standard deviations were nominal.

5. EMPIRICAL EVALUATION

In this section, the size of the tables, the sharing that occurs and the effects of table improvements on size are evaluated using the experimental testbed described in Section 4.

5.1 Size of tables

Figure 5 shows the size of garbage collection tables as a fraction of code size using the best configuration (call sites grouped into sets, a sparse call site table, and a divided stack with stack slots reused for variables with disjoint lifetimes) Code size is the size of generated Java code; code for the runtime system is excluded. The average table size for all 20 benchmarks is 3.6% of code size.

Each bar shows the contribution of various tables. The call site table is the sparse version of the call site table. It includes only every tenth entry in the original table. The auxiliary tables include the back-mapping table and the call site bitmask. Only two programs need escape descriptors: Marmot and impgo. All call sites in the remaining programs can be described with compact descriptors.

There is considerable variation in the size of the descriptor index tables. In part, this is because only 8-bit and 16-bit indexes are supported currently. This causes table sizes to abruptly double when the number of indexes exceeds 256. With more implementation work, indexes whose sizes are not byte multiples could be used.

5.2 The number of indexes is small

The number of indexes relative to the number of call sites is small. Table 2 shows some basic statistics for 201_compress, 202_jess, Marmot, and wc. The first column shows the different sizes of the programs; it gives the executable code size for the Java code. The second column shows the number of call sites. The third column shows the number of indexes. The fourth column shows the number of indexes as a percentage of call sites.

In practice, call sites are grouped into call site sets (see Section 3.1). Even compared to the number of call site sets, the number of indexes is small. The fifth column of Table 2 shows the number of call site sets and the sixth column shows the number of indexes as a percentage of call site sets.

5.3 Most indexes fit into 9 bits

The number of indexes is small enough that for most programs they fit into 8 or 9 bits. Table 2 illustrates this. In fact, of the 20 benchmarks, 13 need 8-bit indexes and 5 need 9-bit indexes. Only two programs need more than 9 bits: 213_javac needs 10-bit indexes and Marmot needs 13-bit indexes.

5.4 Most call sites have few pointers live across them

The number of indexes is small relative to the number of call sites because most call sites have few pointers live across them. These call sites need inordinately few indexes.

The following charts use call site sets to exclude call sites where garbage collection cannot occur. Most call site sets have few pointers live across them. Figure 7 graphs the cumulative distribution of call site sets as a function of the number of pointers live across those sets. There are 3 or fewer pointers live across 90% of the call sites sets for wc, 201_compress, and 202_jess. For Marmot, which is an extreme case, there are 7 or fewer pointers live across 90% of its call site sets. On average, most benchmarks have 4 or fewer pointers live across 90% of their call site sets.

The reuse of indexes is very high for call site sets with only a few pointers live across them. Figure 8 shows the ratio of indexes to call site sets. A low ratio indicates high reuse. As program sizes increase, the reuse improves.

5.5 Call site sets reduce program counter mappings by more than half

Call site sets substantially reduce the size of the mapping from program counter values to descriptor indexes. Table 2 also shows that grouping call sites into sets typically halves the amount of data that must be recorded, reducing both the size of the call site table and the descriptor index table.

5.6 Sparse call site tables use the least amount of space and negligibly increase execution times

Figure 6 compares three different call site table implementations: the original implementation that uses 4-byte entries, the two-level table implementation, and the sparse call site table. As expected, the two-level implementation is roughly half the size of the the original implementation and the sparse call site tables are even better than the two-level implementation. In all cases, call sites are grouped into sets.

Compressing the call site table is important for producing compact tables. A comparison with Figure 5 shows that the original call site table is usually larger than the entire size of the garbage collection tables in the best configuration.

Using a sparse call site table instead of the original call site table negligibly increases overall benchmark execution times. It also has a small effect on garbage collection time unless garbage collection time is very small. Table 3 shows the percentage of running time spent garbage collecting for selected benchmarks when using the original call site table.

Name	Lines of code	Description
		The SPECJVM98 group
201_compress	927	Compression program compressing and decompressing files
202_jess	11K	Java Expert Shell System solving set of puzzles
209_db	1028	An in-memory database program performing a sequence of operations
213_javac	unknown	Java bytecode compiler
222_mpegaudio	unknown	MPEG audio program decompressing audio files in MPEG Layer-3 format
227_mtrt	3753	Multi-threaded ray tracer
228_jack	unknown	Java parser generator generating its own parser
		The Misc Java group
marmot	127K	Marmot compiling _213_javac
cn2	578	CN2 induction algorithm
javacup	8760	JavaCup generating a Java parser
jlex100	14K	JLex generating a lexer for sample.lex, run 100 times.
plasma	648	A constrained plasma field simulation/visualization
slice	989	Viewer for 2D slices of 3D radiology data
SVD	1359	Singular-Value Decomposition (100x600 matrices)
		The IMPACT group
impdes	561	The IMPACT benchmark DES encoding a large file
impgo	32K	The IMPACT transcription of the SPEC95 go.099 benchmark
impgrep	551	The IMPACT transcription of the UNIX grep utility on a large file
impijpeg	8446	The IMPACT transcription of the SPEC95 ijpeg.132 benchmark
docli	8864	The IMPACT transcription of the SPEC95 li.130 benchmark, modified to use the system garbage collector.
impwc	148	The IMPACT transcription of the UNIX wc utility on a large file

Table 1: The Java benchmark programs

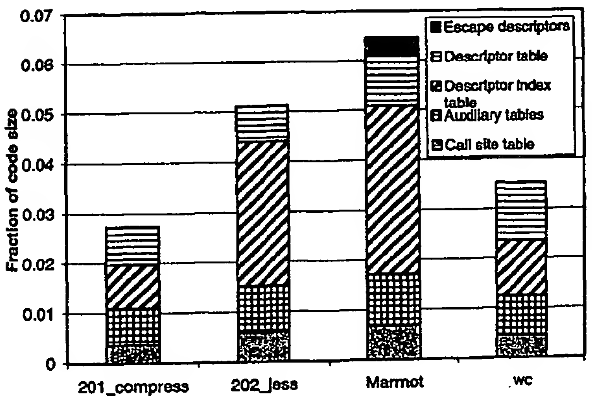

Figure 5: GC table sizes as fractions of code size

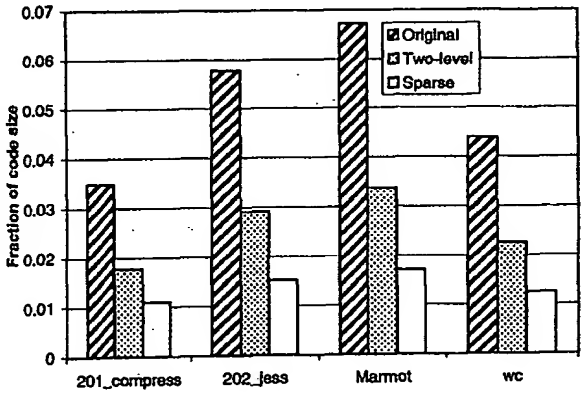

Figure 6: Comparison of call site tables

Program	Code size (bytes)	Call sites	Indexes	Index/CS (%)	Call site sets	Indexes/CSS (%)
201_compress	64,240	1,845	115	6.2	554	20.8
202_jess	199,056	5,520	351	6.4	2,873	12.1
Marmot	2,049,920	61,609	4,972	8.1	34,477	14.5
wc	37,008	941	105	11.1	404	26.3

Table 2: The number of indexes is much smaller than the number of call sites

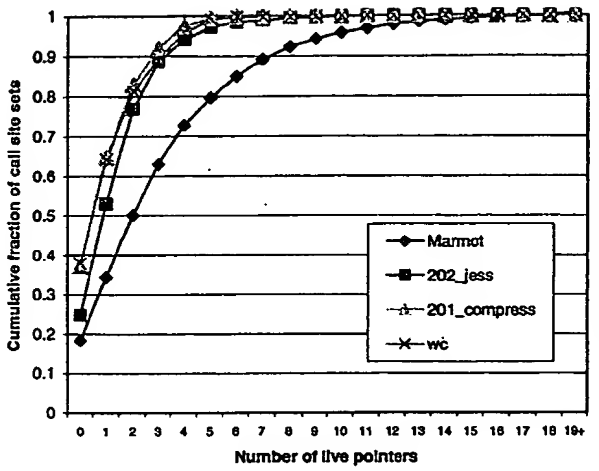

Figure 7: Most call site sets have few pointers live across them

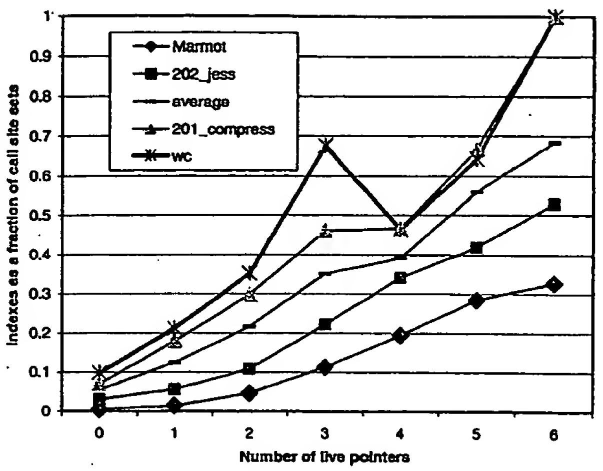

Figure 8: The ratio of indexes to call site sets

It also shows the percentage decrease in the speed of the garbage collector due to the use of sparse call site tables.

The decrease in GC speed for docli is large because stack walking represents a large fraction of the (small) GC time. Docli allocates a large amount of data that dies quickly. There are many collections of the younger generation, each of which includes a stack walk, but very little data survives each collection, so GC time is small.

5.7 Reusing stack slots is important for larger programs

Reusing stack slots for variables with disjoint lifetimes is important for reducing the size of the tables for larger programs. Figure 9 shows the effect of disabling the reuse of stack slots for variables with disjoint lifetimes. The size of tables for Marmot increases by 21%. Other benchmarks affected by this include 213_javac (9%), java_cup (16%), and impgo (35%). The primary cause of the increase is an increase in the size of escape descriptors. A secondary cause is an increase in the number of descriptors, which increases the size of the descriptor table.

5.8 Effect of using an SSB write barrier

A generational collector uses a write barrier to track pointers from older generations to younger generations. This allows younger generations to be collected independently from older generations. A check is placed at each store of a pointer into a heap object. A sequential store buffer (SSB) write barrier appends the locations of cross-generational pointers to a buffer [1, 10, 8]. A garbage collection may be triggered when the buffer overflows.

Thus, using an SSB write barrier introduces additional garbage collection points at pointer stores into heap objects. This increases the size of the tables. Figure 10 shows the effect of this increase. It is primarily due to an increase in the number of call site sets. This increases the size of the tables that map program counter values to indexes.

6. SUMMARY

This paper describes how to produce compact garbage collection tables that are 20-25% of the size of tables produced using previous techniques. The tables share descriptions of pointer locations at many garbage collection points. The sharing is possible because there are relatively few pointers live across most garbage collection points. The size of the descriptions of pointer locations is typically less than 1% of executable code size.

The mapping from program counter values to indexes is represented compactly using several techniques. First, adjacent call sites that have the same live pointer information or cannot garbage collect are grouped into sets. This roughly halves the number of garbage collection points. Second, the table of return addresses is represented as a two-level table with 16-bit offsets or sparsely. The two-level table works well because there are usually many return addresses within a 64K segment of code. The sparse table uses the fact that the executable code contains most of the information in the table. By disassembling the executable code, the exact call site number can be interpolated.

7. REFERENCES

[1] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software: Practice and Experience*, 19(2):171–183, February 1989.

[2] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[3] Standard Performance Evaluation Corporation. SPEC CPU 95 benchmarks. Online version at http://www.spec.org/osg/cpu95, 1995.

[4] Standard Performance Evaluation Corporation. SPEC JVM98 benchmarks. Online version at http://www.spec.org/osg/jvm98, 1998.

[5] Amer Diwan, J. Eliot B. Moss, and Richard L. Hudson. Compiler support for garbage collection in a statically typed language. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 273–282, San Francisco, California, June 1992. SIGPLAN, ACM Press.

[6] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: An

Program	GC time with original table	Decrease in GC speed with sparse table
	(% of running time)	(%)
213_javac	17	0.2
227_mtrt	3	1.3
228_jack	3	2.0
cn2	20	0.2
javacup	13	0.2
docli	0.5	18.9

Table 3: Decrease in garbage collector speed from using a sparse call site set table

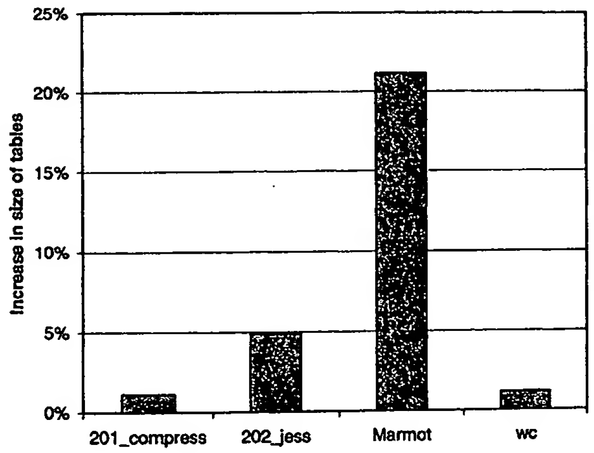

Figure 9: Effect of disabling the reuse of stack slots for variables with disjoint lifetimes on the size of the size of tables

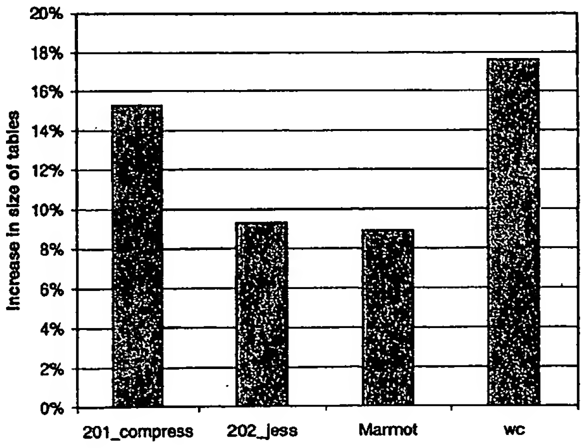

Figure 10: Effect of using an SSB write barrier on tables

optimizing compiler for Java. *Software: Practice and Experience*, 30(3):199–232, March 2000.

[7] David Gay and Bjarne Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *9th International Conference on Compiler Construction (CC'2000)*, Lecture Notes in Computer Science. Springer-Verlag, March 2000. to appear.

[8] Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanović. A comparative performance evaluation of write barrier implementations. In *Proceedings of the ACM '92 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 92–109, Vancouver, British Columbia, October 1992. ACM Press.

[9] C.-H. A. Hsieh, J. C. Gyllenhaal, and W. W. Hwu. Java bytecode to native code translation: the Caffeine prototype and preliminary results. In IEEE, editor, *Proceedings of the 29th annual IEEE/ACM International Symposium on Microarchitecture, December 2–4, 1996, Paris, France*, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. IEEE Computer Society Press.

[10] Richard L. Hudson, J. Eliot B. Moss, Amer Diwan, and Christopher F. Weight. A language-independent garbage collector toolkit. COINS Technical Report 91-47, University of Massachusetts, Amherst, June 1991.

[11] Erik Ruf. Removing synchronization operations from Java. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, Vancouver, Canada, June 2000. ACM Press. to appear.

[12] James M. Stichnoth, Guei-Yuan Lueh, and Michal Cierniak. Support for garbage collection at every instruction in a java compiler. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 118–127, May 1999.